Architectural and Compiler Support for Energy Reduction in the Memory Hierarchy of High Performance Microprocessors*

Nikolaos Bellas Ibrahim Hajj, Constantine Polychronopoulos and George Stamoulis†
Department of Electrical & Computer Engineering
and the Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 West Main Street, Urbana, IL 61801

Abstract

In this paper we propose a technique that uses an additional mini cache located between the I-Cache and the CPU core, and buffers instructions that are nested within loops and are continuously otherwise fetched from the I-Cache. This mechanism is combined with code modifications, through the compiler, that greatly simplify the required hardware, eliminate unnecessary instruction fetching, and consequently reduce signal switching activity and the dissipated energy.

We show that the additional cache, dubbed *L-Cache*, is much smaller and simpler than the I-Cache when the compiler assumes the role of allocating instructions in it. Through simulation, we show that, for the SPECfp95 benchmarks, the I-Cache remains disabled most of the time, and the "cheaper" extra cache is used instead. We present experimental results that validate the effectiveness of this technique, and present the energy gains for most of the SPEC95 benchmarks.

1 Introduction

The problem of the wasted energy caused by unnecessary activity in various parts of the CPU during code execution has traditionally been ignored in code optimization and architecture design. Processor architects and compiler writers are concerned with system performance/throughput and they do little, if anything at all, to eliminate energy/power dissipation at this level. Researchers in the CAD community have started tackling the problem of power minimization through compiler transformations, yet this process is still in its infancy.

An increasing number of architecture features have been exposed to the compiler to enhance performance. The advantage of this cooperation is that the compiler can generate code that exploits the characteristics of the machine and avoids expensive stalls. We believe that such schemes can also be applied for power/energy optimization by exposing the memory hierarchy features in the compiler.

We are targeting the activity caused by the I-Cache subsystem which is one of the main power consumers in most of today's microprocessors. The on-chip L1 and L2 caches of the 21164 DEC Alpha chip dissipate 25% of the total power of the processor [1]. In [2], a power analysis of the DLX processor shows that the I-Cache memory and the I-Cache controller are responsible for almost 50% of the total power consumption for some programs. The StrongARM SA-110 processor from DEC, which targets specifically low power applications, dissipates about 27% of the power in the I-Cache.

The paper is organized as follows: In section 2, we review related work. In section 3, we outline our approach and give some motivation. In section 4 we present the steps proposed for compiler enhancements, and in section 5 we describe the hardware modifications. Power estimation and experimental evaluations are given in sections 6 and 7, respectively. Section 8 gives conclusions.

2 Related Work

There has been little research done in the field of software-based power minimization. In [3][4], a brief review of some compiler techniques that are of interest in the power minimization arena is presented. As expected, standard compiler optimizations, such as loop unrolling, software pipelining etc., are also beneficial for the reduction of energy since they reduce the running time of the code. In [5] and in subsequent papers, a methodology that attempts to relate the power consumed by a microprocessor to the software that executes on it, is described. This is different from the often used "bottom-up" approach in which power models are built using a layout, gate or RT-level model of each unit and the power consumption of the whole chip is the sum of the power consumed by each component unit. The authors characterize each instruction of a given microprocessor in terms of the power it dissipates when it is executed.

The problem of register allocation, which is central in the code generation phase of a compiler, is solved aiming at the minimization of switching activity in [6][7]. In [7], the problem of optimizing the energy for the variable allocation in registers and memory is solved using a minimum cost network flow.

The impact of memory hierarchy in minimizing power consumption, and the exploration of data-reuse so that the power required to read or write data in the memory is reduced is addressed in [8]. The same authors propose

*This work was supported by Intel Corp., Santa Clara, CA

†Intel Corporation, Santa Clara, CA

ISLPED 98, August 10-12, 1998, Monterey, CA USA
© 2000 ACM ISBN 1-58113-059-7/98/08...$5.00

a novel way to organize complex data structures in the memory hierarchy, so that a cost function is minimized. The goal here is to reduce power when complex data structures are manipulated in various ways [9]. In [10], a mechanism is described, which enables the by-pass of the I-Cache by storing the instructions of a loop in an extra buffer. That method makes a series of restrictive assumptions such as all the basic blocks within a loop are executed upon entry in the loop, and all the loops are small enough to be accomodated in the extra buffer. Our method is applicable to any type of code without any restrictions.

In [11], an extra, smaller cache is added between the CPU and the L1 caches. The extra caches, called filter caches in [11], deliver large energy gains at the expense of large miss rates. That scheme is better suited for embedded applications which can trade off energy reduction for performance degradation, but is probably unacceptable for the high-end processor market. Our method also uses an extra cache, but it almost eliminates the performance overhead by having the compiler exploit the new memory hierarchy, and eliminate unnecessary misses in the extra cache.

3 Motivation and Approach

During a loop execution, the I-Cache unit frequently repeats its previous tasks over and over again: if a program is caught in a loop, the I-Cache unit fetches the same instructions to the CPU core, and the ID decodes the very same instructions. This approach works for performance but it unnecessarily performs more work, and thus it dissipates more power than really needed.

All the instructions that belong to a frequently executed loop can be fetched only the first time the thread of control passes through them. Subsequently, they can be stored in a special internal cache (the L-Cache) which is placed between the I-Cache and the CPU core. Each time the IF unit attempts to fetch an instruction from within the loop, the instruction that resides in this cache can be used instead. In the ideal case, the I-Cache unit can be shut down for the duration of the loop, as it does not need to operate, and its energy dissipation can be saved.

The approach advocated in our scheme relies on the use of profile data from previous runs to select the best instructions to be cached. The unit of allocation is the basic block, i.e. an instruction is placed in the L-Cache only if it belongs to a selected basic block. After selection, the compiler lays out the target program so that the selected blocks are placed contiguously before the non-placed ones. The main effort of the compiler focuses on placing the selected basic blocks in positions so that two blocks that need to be in the L-Cache at the same time, do not map in the same L-Cache location.

The compiler maximizes the number of basic blocks that can be placed in the L-Cache by determining their nesting, and using their execution profile. A basic block, as opposed to the whole loop, is the unit of allocation in the L-Cache because, in most cases, the loop contains basic blocks which are seldom executed during typical runs. These are blocks that take care of an exception condition or do error handling. If the whole loop was to be allocated in the L-Cache, these basic blocks would occupy space but hardly ever used.

4 Compiler Enhancements

The first technique we are using is function inlining. Inlining replaces the function call with the body of the called function [12] [13]. This step aims at exposing as many basic blocks as possible in frequently executed routines. Our scheme assumes that no interprocedural basic block allocation can take place, i.e., at any given time, only basic blocks that belong to the same function can reside in the L-Cache. This precaution is taken since the compiler cannot know a-priori where the linker/loader will place the functions in the memory address space. Hence, each function in the source code is considered separately. Only functions which contribute a large number of execution cycles are selected for inlining.

After inlining, the code is laid out so that the most frequently executed basic blocks are placed in the L-Cache. In order to do that, we need to place these blocks contiguously in memory so that they do not overlap. Consider the following code :

```
do 100 i=1, n
  B1;      # basic block
  if (error) then
     error handling;
  B2;      # basic block
100 continue
```

When the code is compiled, the basic blocks B1 and B2 will be separated by the code for the if-statement in the final layout. If the L-Cache size is smaller than the sum of the sizes of B1, B2 and the if-statement, but larger than the sum of the sizes of B1 and B2, the blocks B1 and B2 will overlap when stored in the L-Cache. Therefore, we need to place B1 and B2 one after the other and leave the if-statement at the end.

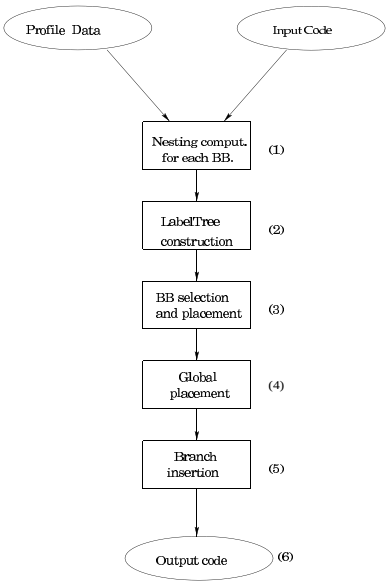

Figure 1: Block placement overview

This is usually the case in loops. Blocks that are executed for every iteration are intermingled with blocks that are seldom executed. We identify such cases and move the infrequently executed code away so that the normal flow of control is in a straight-line sequence. This entails the insertion of extra branch and jump instructions to retain the original semantics of the code. The algorithm is outlined in Fig. 1.

The object code and profile data for the original program are used as input to our tool. The output

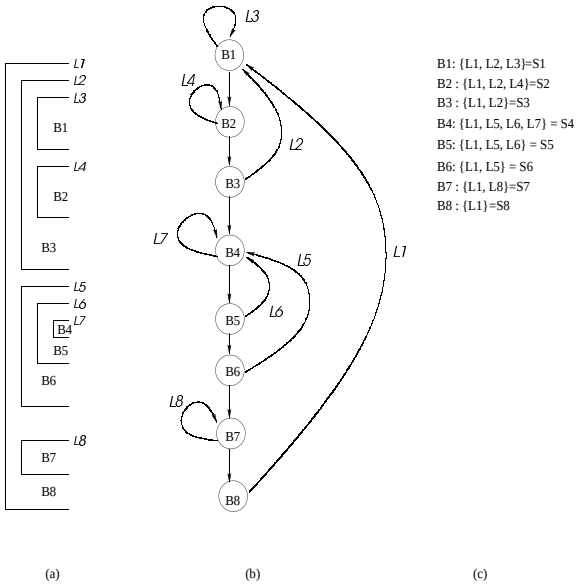

Figure 2: First step of block placement

produced is an equivalent object code in which some of the basic blocks have been reordered and placed in specific memory locations.

The control flow graph is built for each function of the original program in step (1). Note that the program can be either the original or the one that has been created after inlining.

Next, in the same step, the tool finds the loops and the nesting for every basic block [14]. Basic blocks within loops that contain function calls are not considered for placement. A $LabelSet(B)$ for every basic block B is the set of loops to which B belongs. If B is not nested, $LabelSet(B) = \{\}$. If B is enclosed in loops L_1, L_2 and L_3, then $LabelSet(B) = \{L_1, L_2, L_3\}$. These are the same sets used in [15]. In Fig. 2, an example is given to describe the data structures used, and the information produced during the first step of the algorithm. A loop nesting is shown in 2(a), the corresponding CFG in 2(b), and the LabelSets in 2(c).

In step (2), we construct a tree using the LabelSets as follows: the nodes are the different LabelSets found in the previous step. There is an arc between two such LabelSets $< l_1, l_2 >$ if l_1 is a proper subset of l_2 (Fig. 3a). This data structure is a tree, called $LabelTree$, since each basic block cannot have two different nestings.

Step (3) (shown in Fig. 4) takes over the main part of the allocation algorithm.

The algorithm scans the basic blocks in descending order of execution frequency. Hence, the most important blocks are the first to be considered and have a greater chance to be placed in the L-Cache. For every node in the LabelTree we designate a size, which denotes the position in the L-Cache where a basic block of the node should be placed in every step of the algorithm. The size should always be less than or equal to the cache size or the current basic block cannot be placed in the L-Cache.

The first step is to propagate the effect of the size of the basic block under consideration towards the leaves of the tree rooted at node N (DOWN_TRAV()). Suppose, for example, that the current basic block is B_3

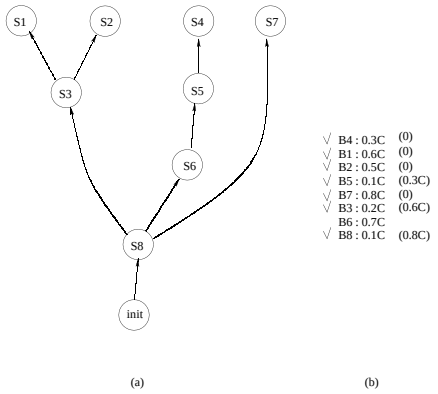

Figure 3: LabelTree

in Fig. 3a. Both nodes B_1 and B_2 have already been considered and placed in the L-Cache. The size of B_3 added to the $max(size(B_1), size(B_2))$ should not exceed the cache size C. If this is the case, B_3 is placed in the L-Cache. In other words, B_3 will remain in the L-Cache while B_1 and B_2 are executed, and it will not be replaced. This step aims at placing B_3 in a different cache position from both B_1 and B_2. If B_3 overlapped with them, it would have to be fetched from the I-Cache instead, since it would be replaced by B_1 after being executed. This technique maximizes the number of basic blocks that are placed in the cache and avoids conflicts between them.

If $max(size(B_1), size(B_2)) + size(B_3) > C$, the placement of B_3 is not possible, and the algorithm continues with the next basic block.

Subsequently, the algorithm calls UP_TRAV() which propagates the effect of the new placement to the outer blocks. This, in effect, reduces the chance of the outer blocks to be placed in the L-Cache, which is not bothering at all, since we are mostly interested for the inner, most frequently executed blocks. In Fig. 3a, the annotated LabelTree for the example in Fig. 2 is given with the final placement of the basic blocks in 3b. All the blocks except B_6 are placed in the L-Cache (the positions are in the parentheses–C is the size of the L-Cache).

The algorithm is greedy, because it tries to accumulate as many important basic blocks as possible in the L-Cache. In the case where the most frequently executed basic blocks are the most deeply nested, the algorithm will succeed in putting all of them in the L-Cache provided that the size of each one is smaller than the cache size.

A basic block will not be selected for placement in algorithm Allocate() if any of the following is true:

- It belongs to a library and not to a user function. We follow the convention that only user functions are candidates for placement since they have the tightest and deepest loops.
- The algorithm finds that the basic block was too large to fit in the L-Cache. This can be either because the size of the block is larger than the cache size, or because it cannot fit at the same time with other, more important, basic blocks.
- Its execution frequency is smaller than a threshold, and is thus deemed unimportant.
- It is not nested in a loop. There is no point in placing such a basic block in the L-Cache since it

```
1. void Allocate(LabelTree T, CacheSize C) /* T root of the LabelTree, C size of L-Cache */
2.    for every node N in T set size(N) = 0;
3.    for every basic block BB in the program in descend. order of number of times executed do
4.       let N be the node in T that BB corresponds to;  /* N is unique */
5.       if N is the root next;  /* we don't place BB which are not nested */
6.       old_size(N) = size(N);
7.       fit = DOWN_TRAV(N, BB, C);
8.       if (fit == FALSE) then continue;   /* next basic block */
9.       UP_TRAV(N);
10.      put the basic block BB in the L-Cache in position old_size(N);
11.   end for;
12. end Allocate;

13. boolean DOWN_TRAV(TreeNode N, BasicBlock BB, CacheSize C)
14.    if (DOWN_TRAV_FIRST(N, BB, C) then
15.       DOWN_TRAV_SEC(N);
15.       return true;
16.    else
17.       return false;
18. end DOWN_TRAV;

19. flag = true;
20. boolean DOWN_TRAV_FIRST(TreeNode N, BasicBlock BB, CacheSize C)
21.    if (N != NULL and flag) then
22.       if (size(N) + size(BB) > C) then
23.          flag = FALSE;
24.       else
25.          temp_size(N) = size(N) + size(BB);
26.       end if;
27.       for all children of N do
28.          DOWNTRAV_FIRST(N->child);
29.    end if;
30. end DOWN_TRAV_FIRST;
31. void DOWN_TRAV_SEC(TreeNode N)
32.    if (N != NULL) then
33.       size(N) = temp_size(N);
34.       for all children of N do
35.          DOWN_TRAV_SEC(N->child);
36.    end if;
37. end DOWN_TRAV_SEC;

38. procedure UP_TRAV(TreeNode )
39.    while (N->parent) do
40.       N = N->parent;
41.       size(N) = max(size(N->child)) among all children;
42.    end while;
43. end UP_TRAV;
```

Figure 4: Placement Algorithm

will be executed only once for each invocation of its function.

- Even if its execution frequency is large, its *execution density* might be small. For example, a basic block that is located in a function which is invoked a lot of times might have a large execution frequency, but it might only be executed few times for every function invocation. We define the execution density of a basic block as the ratio of the number of times it is executed to the number of times that the function in which it belongs, is invoked.
- Finally, a very small basic block is not placed in the L-Cache even if it passes all the other requirements. The extra branch instructions that might be needed to link it to its successor basic blocks will be an important overhead in this case.

A basic block is placed in the L-Cache only if it is expected to stay there for a long period of time without getting replaced. This, in effect, decouples the communication between the I-Cache and the L-Cache, and reduces the traffic between them.

Step (4) in our methodology is the placement of the basic blocks in the global address space. The algorithm takes as input the placement of the basic blocks with respect to the L-Cache and tries to minimize the necessary space as much as possible.

Finally, in step (5), the tool performs the actual layout of the code, and restructures the CFG. It starts by

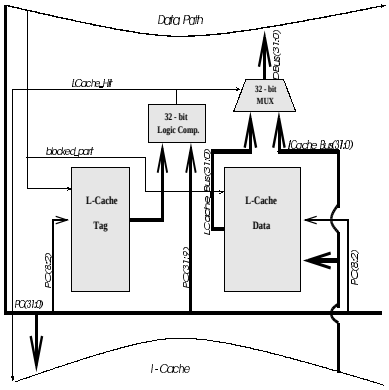

Figure 5: LCache organization

placing the basic blocks that were selected in step (3). Those blocks are placed at the beginning, in the memory locations assigned to them in step (4). Then, all the other basic blocks are placed contiguously. Branches are placed at the end of the blocks, if needed, to sustain the functionality of the code. These branches introduce a performance overhead with respect to the initial code. As we will see in the experimental evaluation section, the overhead is small.

5 Hardware modifications

The extra hardware needed to implement our scheme is shown in Fig. 5.

The functionality of the L-Cache is as follows: The *Program Counter (PC)* is presented to the L-Cache tag at the beginning of the clock cycle. The L-Cache tag will only be enabled if the *"blocked_part"* signal is on. This signal is generated by the Instruction Fetch Unit (IFU), and its meaning is explained in the following paragraphs. In that case, the comparator checks for a match, and if it finds one, it instructs the multiplexer to drive the contents of the L-Cache in the data path. The I-Cache is disabled.

In case of a L-Cache miss (*"LCache_Hit"* is off), the I-Cache controller activates the I-Cache at the next clock cycle, and de-activates the data portion of the L-Cache. We should emphasize at this point, that the L-Cache and the I-Cache are not accessed in parallel. The I-Cache is accessed if the L-Cache misses using an extra clock cycle, whereas the L-Cache is accessed only when *"blocked_part"* = on. If *"blocked_part"* = off, the I-Cache controller activates the I-Cache without waiting for the *"LCache_Hit"* signal.

Recall that the compiler has already laid out the code so that the basic blocks that are destined for the L-Cache are placed before the others. A 32-bit register is used to hold the address of the first non-placed block in the main memory layout. If the PC has a value less than that address, the 32-bit comparator will set *"blocked_part"* = on, else this signal will be set to off. This way, the machine can figure out which portion of the code executes with only an extra comparison.

This simplification is only possible because of the way that the code has been restructured in the compilation phase. Notice also that if *"blocked_part"* = on, the L-Cache can still miss: this will happen, for example, when the basic block to be placed in the L-Cache has not been executed before, i.e. the first time the thread of control passes through it. Therefore, the tag portion

Benchmark	*L-Cache size*					
	32 instr.	64 instr.	128 instr.	256 instr.	512 instr.	1024 instr.
tomcatv	15.25%	39.97%	43.01%	99.84%	99.94%	99.94%
swim	0.10%	99.67%	99.88%	99.93%	99.93%	99.93%
su2cor	10.05%	42.99%	92.55%	98.17%	99.22%	99.22%
hydro2d	22.61%	74.09%	93.25%	94.48%	94.49%	94.49%
mgrid	5.39%	7.31%	97.79%	99.38%	99.38%	99.38%
applu	12.62%	43.12%	66.87%	73.01%	73.35%	84.28%
turb3d	3.89%	50.90%	74.44%	74.87%	77.90%	77.90%
apsi	10.50%	34.83%	79.65%	83.84%	86.76%	86.76%
wave5	25.30%	44.54%	70.35%	94.11%	96.16%	96.16%
go	1.36%	1.36%	1.36%	1.36%	1.36%	1.36%
m88ksim	46.16%	46.16%	46.16%	46.16%	46.16%	46.16%
compress95	15.30%	15.30%	15.30%	15.30%	15.30%	15.30%
li	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%

Table 1: L-Cache utilization statistics : percent of instructions fetched from the L-Cache

of the L-Cache is still needed. In that case, the I-Cache will provide the new instruction to the data path, as well as to the L-Cache.

The block size of the L-Cache is one word. This simplifies the transfer of data from the I-Cache to the L-Cache in case of an L-Cache miss. Since our algorithm selects a basic block only if it is expected to reside in the L-Cache for a long period of time, the one-word block size does not severely affect the L-Cache hit rates.

Finally, we extend the instruction set, and we add a new instruction called *"alloc"* which marks the boundary between the placed and the non-placed code. This extra instructions is used to store the address of the first non-placed instruction in the 32-bit register, as described above, and is the first instruction to be executed upon entry in a procedure.

6 Power Estimation

We have deceloped our cache energy models based on [16]. It is a transistor level model which uses the run-time characteristics of the cache to estimate the energy dissipation of its main components. A 0.8um technology with 3.3 Volts power supply is assumed. The cache energy is a function of the cache complexity (cache size, block size, and associativity), its internal organization (banking), and the run time statistics (mumber of accesses, hits, misses, average number of bits read, input switching probabilities). The model is used for both the I-Cache and the L-Cache.

The run-time statistics are extracted by simulating traces of the SPEC95 benchmarks in a cache simulator. We used the SpeedShop [17] set of tools from SGI to gather the profile data and generate the dynamic instruction traces that were fed into our cache simulator.

7 Experimental Evaluation

The SPEC95 benchmarks were compiled with the MIPS compiler. First, we ran the benchmarks to collect the profile data. The data were used to drive the inline and the block placement heuristics. The tool, along with the restructuring of the body of the program, selected various statistics regarding the quality of the generated code. The utilization statistics of the L-Cache are given in Table 1. The percentage of the dynamically executed instructions that are taken from the L-Cache is given for every benchmark, and for 6 different sizes of the L-Cache. An L-Cache with capacity n instr. has size $4n$ bytes. The I-Cache is 16KB, direct-mapped, with a block size of 32 bytes. A basic clock was selected for placement only if it had execution time larger than 0.01% of the execution time of the program, had at least five instructions, and an execution density of at least five.

Benchmark	*L-Cache size*	
	64 instr.	128 instr.
tomcatv	0.2%	0.0%
swim	0.0%	0.0%
su2cor	0.1%	1.0%
hydro2d	3.8%	5.2%
mgrid	0.9%	0.9%
applu	8.0%	24.0%
turb3d	1.4%	3.7%
apsi	2.2%	4.0%
wave5	4.0%	2.7%
FP aver.	2.3%	4.6%
go	1.6%	1.9%
m88ksim	2.3%	2.3%
compress	2.3%	2.3%
li	0.0%	0.0%
INT aver.	1.5%	1.5%

Table 2: Execution time overhead

This percentage is high for all the SPECfp95 benchmarks, reflecting the efficacy of our approach for these programs. As expected, a larger L-Cache is more successful in storing basic blocks and therefore in disabling the I-Cache for a larger period of time. In some cases, even a small L-Cache is capable of effectively shutting-down the I-Cache for the duration of the program execution. The law of diminishing returns applies here as well, since a very large L-Cache (1024 instr.) is usually as succesful as smaller ones. In most cases, a 256 instr. L-Cache approximates the performance of an infinite size L-Cache.

On the other hand, most integer benchmarks do not have a large number of basic blocks that can be cached in the L-Cache. They are also insensitive to the cache size variation, which is to be expected since the basic blocks of integer programs are generally small. Most of the basic blocks of the SPECint95 benchmarks are not nested, or, they are nested within a loop that contains a function call; hence, they cannot be included in the L-Cache.

The execution time overhead is given in Table 2 only for L-Caches with size 256 and 512 bytes. The overhead is due to the insertion of jump instructions at the end of the basic blocks which have been selected from the algorithm, and the non-perfect hit rate in the L-Cache. The overhead has maximum value 24%, but is usually less than 4.0%.

Finally, the energy savings are depicted in Fig. 6.

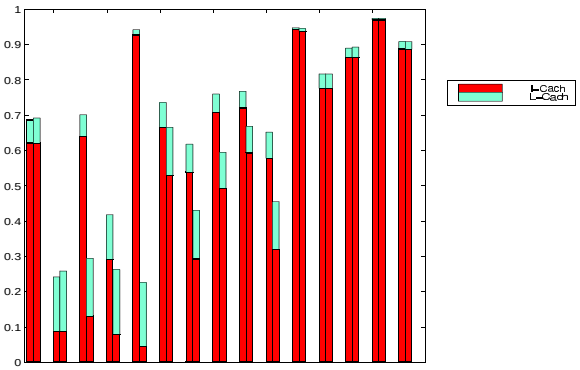

Figure 6: Normalized energy savings for SPEC95 benchmarks

The energy dissipation of the modified scheme (L-Cache and I-Cache) is normalized with respect to the energy dissipation of the original scheme (only I-Cache). The 2 bars represent the savings for the schemes with L-Caches of 256 and 512 bytes, respectively. In both graphs, the original scheme has an energy dissipation of 1.

In the modified scheme, both caches will contribute to the total energy dissipation. Their individual contribution is shown with dark color for the I-Cache and light color for the L-Cache. The sum of the individual contributions is the total energy consumption of the modified scheme with respect to the original. For example, *su2cor* has a energy consumption which is around 30% of that of the original scheme when the L-Cache is 128 instr. large. This is due to the L-Cache (15%) and the I-Cache (15%).

An optimal L-Cache has size of 128 instr. (i.e. 0.5 KB) for the FP benchmarks. Small caches are not very succesful in disabling the I-Cache. Larger caches, on the other hand, have larger energy dissipation per access, and do not accomodate many more basic blocks than average sized ones. The energy dissipation drops as the size increases, but it goes up again for the larger caches. On the average, the new scheme dissipates only 45% of the energy of the original scheme for the FP benchmarks, when a 128 instr. L-Cache is included. Notice also that the new scheme never dissipates more energy than the original one.

8 Conclusions

This paper presented a paradigm for hardware/compiler co-design that targets activity minimization in a processor. These techniques are orthogonal to the standard circuit or gate level techniques that are traditionally used by designers to reduce energy and can therefore be used to further reduce energy consumption without impairing performance. This paradigm describes a more judicious use of the I-Cache unit of a processor when the flow of control is caught within a loop. The compiler is given the responsibility to restructure the code. The aim is to minimize the overlap between basic blocks that are selected to be placed in an extra cache.

The gains from this modifications can be very important for machines with a very high energy consumption in the I-Cache unit. The savings are dependent on the structure of the program, and can be maximized for scientific computations with regular loop patterns.

References

[1] J. Edmondon, "Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor," *Digital Technical Journal*, vol. 7, no. 1, pp. 119–135, 1995.

[2] A. Kalambur and M. J. Irwin, "An Extended Addressing Mode For Low Power," in *International Symposium of Low Power Electronics and Design*, pp. 208–213, IEEE/ACM, Aug. 1997.

[3] V. Tiwari, S. Malik, and A. Wolfe, "Compilation Techniques for Low Energy: An Overview," in *Proceedings of the IEEE Symposium on Low Power Electronics*, (San Diego, CA), Oct. 1994.

[4] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh, "Techniques for Low Energy Software," in *International Symposium of Low Power Electronics and Design*, pp. 72–75, IEEE/ACM, Aug. 1997.

[5] V. Tiwari, S. Malik, and A. Wolfe, "Power Analysis of Embedded Software: A First Step Towards Software Power Minimization," *IEEE Transactions on VLSI Systems*, vol. 2, pp. 437–445, Dec. 1994.

[6] J. Chang and M. Pedram, "Register Allocation and Binding for Low Power," in *Design Automation Conference*, pp. 29–35, IEEE/ACM, 1995.

[7] C. Gebotys, "Low Energy Memory and Register Allocation Using Network Flow," in *Design Automation Conference*, pp. 435–440, IEEE/ACM, June 1997.

[8] J. Diguet, S. Wuytack, F. Catthoor, and H. D. Man, "Formalized Methodology for Data Reuse Exploration in Hierarchical Memory Mappings," in *International Symposium of Low Power Electronics and Design*, pp. 30–35, IEEE/ACM, Aug. 1997.

[9] S. Wuytack, F. Catthoor, and H. DeMan, "Transforming Set Data Types to Power Optimal Data Structures," *IEEE Transcactions on Computer-Aided Design*, vol. 15, pp. 619–629, June 1996.

[10] R. Bajwa, M. Hiraki, H. Kojima, D. Gorny, K. Nitta, A. Shridhar, K. Seki, and K. Sasaki, "Instruction Buffering to Reduce Power in Processors for Signal Processing," *IEEE Transactions on VLSI Systems*, vol. 5, pp. 417–424, Dec. 1997.

[11] Johnson Kin, Munish Gupta and William Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure," in *IEEE International Symposium on Microarchitecture*, pp. 184–193, Dec. 1997.

[12] S. McFarling, "Procedure Merging with Instruction Caches," in *Conference on Programming Language Design and Implementation*, pp. 71–79, ACM SIGPLAN, June 1991.

[13] A. Ayers, R. Gottlieb, and R. Schooler, "Aggressive Inlining," in *Conference on Programming Language Design and Implementation*, pp. 134–145, ACM SIGPLAN, June 1997.

[14] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools*. Addison–Wesley, 1986.

[15] S. McFarling, "Program Optimization for Instruction Caches," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 16–27, ACM SIGPLAN, June 1989.

[16] S. Wilson and N. Jouppi, "An Enhanced Access and Cycle Time Model for On-Chip Caches," DEC WRL Technical Report 93/5, July 1994.

[17] *SpeedShop User's Guide*. Silicon Graphics Inc., 1996.